Sliding-Window Top-k Queries on Uncertain Streams

Cheqing Jin[1] Ke Yi[2] Lei Chen[2] Jeffrey Xu Yu[3] Xuemin Lin[4,5]

[1]East China University of Science and Technology, Shanghai, China
cqjin@ecust.edu.cn
[2]Hong Kong University of Science and Technology, Hong Kong, China
{yike,leichen}@cse.ust.hk
[3]The Chinese University of Hong Kong, Hong Kong, China
yu@se.cuhk.edu.hk
[4]The University of New South Wales, Sydney, Australia
[5]National ICT Australia, Sydney, Australia
lxue@cse.unsw.edu.au

ABSTRACT

Query processing on uncertain data streams has attracted a lot of attentions lately, due to the imprecise nature in the data generated from a variety of streaming applications, such as readings from a sensor network. However, all of the existing works on uncertain data streams study unbounded streams. This paper takes the first step towards the important and challenging problem of answering sliding-window queries on uncertain data streams, with a focus on arguably one of the most important types of queries—top-k queries.

The challenge of answering sliding-window top-k queries on uncertain data streams stems from the strict space and time requirements of processing both arriving and expiring tuples in high-speed streams, combined with the difficulty of coping with the exponential blowup in the number of possible worlds induced by the uncertain data model. In this paper, we design a unified framework for processing sliding-window top-k queries on uncertain streams. We show that all the existing top-k definitions in the literature can be plugged into our framework, resulting in several succinct synopses that use space much smaller than the window size, while are also highly efficient in terms of processing time. In addition to the theoretical space and time bounds that we prove for these synopses, we also present a thorough experimental report to verify their practical efficiency on both synthetic and real data.

1. INTRODUCTION

It has become an important issue to process uncertain (probabilistic) data in many applications, such as sensor networks, data cleaning, and objects tracking. For a given uncertain dataset, there are many possible instances called worlds, and the *possible worlds* semantics has been widely used [12, 20, 28, 29, 30, 33].

Consider a radar-controlled traffic monitoring application, where a radar is used to detect car speeds with possible errors caused by nearby high voltage lines, close cars' interference, human operators mistakes, etc. It implies that a speed reading is correct with certain probability. Table 1 shows a simple uncertain dataset for car speed readings. For example, the 4th record indicates that a Benz car (No. W-541) runs at speed 2 ($\times 10$) km per hour through the monitoring area at AM 10:38 with probability 0.4.

ID	Reading Info	Speed ($\times 10$)	prob.
1	AM 10:33, Honda, X-123	5	0.8
2	AM 10:35, Toyota, Y-245	6	0.5
3	AM 10:37, Mazda, Z-341	8	0.4
4	AM 10:38, Benz, W-541	2	0.4

Table 1: 4 Radar reading records

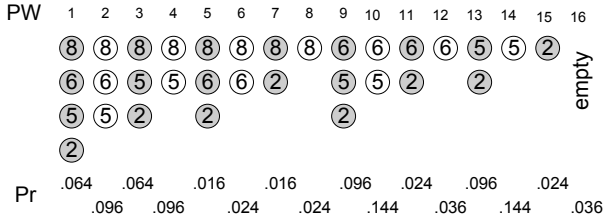


Figure 1: The possible worlds at time 4

For the 4-tuple uncertain dataset given in Table 1, there are in total 16 possible worlds for all the 4 speed readings: 8, 6, 5, and 2. Here, a possible world is a set of speed readings associated with a probability of the set, which is computed based on both the existence of all the tuples in the possible world and the absence of all the tuples in the dataset that are not in the possible world, assuming mutual independence among the tuples. Figure 1 shows all the 16 possible worlds. In Figure 1, the top line numbers all the 16 possible worlds; a possible world is of a subset of the 4 speed readings, represented in the middle, and is associated with an occurring probability of the possible world below. Consider the 10th possible world that contains a set of 2 speed readings: 6 and 5. The probabilities of the existences of 6 and 5 are 0.5 and 0.8, respectively, as given in Table 1. The probabilities of the absence of 8 and 2 are both $1 - 0.4$. Therefore, the probability of the 10th possible world becomes 0.144 ($= 0.5 \times 0.8 \times (1 - 0.4) \times (1 - 0.4)$).

Uncertain data streams. In many real application scenarios, the collected uncertain data is returned in a streaming fashion, such as the radar readings example in Table 1 and the collected sensor readings from a real-time monitoring sensor network. These uncertain data streams have attracted a lot of attention very recently [1, 9, 22, 23, 34]. Since large amounts of such streaming data could arrive rapidly, the goal here is to design both space- and time-efficient query processing techniques. On the other hand, streaming data is

VLDB '08, August 24-30, 2008, Auckland, New Zealand

also highly time-sensitive: each item arrives with a timestamp, and people are generally more interested in the recent tuples than those in the far past. There are two models for dealing with the time aspect on data streams. One is the so-called *time-decaying* model, which assigns a weight to each tuple that is exponentially decreasing over time. This model usually works together with statistical aggregates [10, 11], such as averages, histograms, heavy hitters, etc., but may not be well defined for many other general database queries such as top-k queries. The other model is the more popular *sliding window* model, where we are interested in evaluating the query on tuples that have arrived in, say, the last 24 hours. This model is more general, since any query defined on a static dataset can be also defined with respect to a sliding window. In addition, sliding-window queries are usually required to be *continuous*, i.e., the user should be alerted whenever the query result changes, so that he/she always has the up-to-date query result for the current sliding window.

Although query processing in sliding windows has been thoroughly studied on certain data streams (see [17] and the references therein), sliding-window queries on uncertain streams are still an untapped territory, due to the many challenges brought by the strict space and time requirements of processing both arriving and expiring tuples in the high-speed stream, combined with the difficulty of coping with the exponential blowup in the number of possible worlds induced by the uncertain nature of the data. Previous works on uncertain streams [9, 22, 23, 34] only deal with unbounded streams but not sliding windows. In this paper, we make a first step towards answering sliding-window queries on uncertain streams, focusing on arguably one of the most important types of queries—top-k queries.

Top-k queries on uncertain data. Top-k queries have been recently studied in the setting of uncertain data. Given a ranking function, the goal is to find the top-k ranked tuples in a given uncertain dataset. Soliman et al. [30] defined two types of top-k queries over a uncertain dataset, called U-Top$k$ and U-kRanks. Hua et al. [20] defined a probabilistic threshold top-k query, denoted PT-k. We introduce them in brief below.

The U-Top$k$ query returns the top-k tuples in all possible worlds with maximum probability. Let $k = 2$, the query U-Top$k$ upon the uncertain dataset (Table 1) returns $\{6, 5\}$, because this vector is ranked top in the 9th and 10th possible worlds, with probability 0.24 ($= 0.096 + 0.144$). The probability 0.24 is higher than that of any other two speed readings. For example, it is higher than the probability of having 8 and 6 as the top-2, whose probability is 0.2 ($= 0.064 + 0.096 + 0.016 + 0.024$), as 8 and 6 are ranked top-2 in the 1th, 2nd, 5th, and 6th possible worlds.

The U-kRanks query returns the winner for the i-th rank for all $1 \leq i \leq k$. Consider the same example. When $k = 2$, the U-kRanks query upon Table 1 returns $\{8, 5\}$, because the probability of having 8 as the winner in the first rank is higher than any other speed readings, and the probability of having 5 as the winner of the second rank is higher than any other speed readings.

The PT-k query returns all the tuples with maximum aggregate probability greater than a user-given threshold p, where the aggregate probability represents the sum of probabilities of ranking as one of the top-k ranked tuples in all possible worlds. Consider Table 1, the answer is $\{5, 6, 8\}$, with probability 0.64, 0.5, and 0.4, respectively (assuming $p = 0.3$). The speed reading 5 is ranked in top-2 in the 3rd, 4th, 9th, 10th, 13th, and 14th possible worlds, with the aggregate probability 0.64 ($= 0.064 + 0.096 + 0.096 + 0.144 + 0.096 + 0.144$).

The three top-k query definitions have different semantics and may possibly give different results on the same uncertain dataset. The intention of U-Top$k$ is to find the most likely top-k ranking list in a random possible world, and to preserve such ranking order. U-kRanks considers the winner in every individual rank, and PT-k considers the probability of being one of the top-k. It is not the focus of this paper to argue which definition is better than the others, or to propose yet another definition. Indeed, the particular choice should probably be application-dependent. Our goal, on the other hand, is to design a unified framework for processing sliding-window top-k queries, which can be coupled with any of the definitions above.

To make the presentation concise, we use the PT-k query to illustrate our framework; discussions on the other top-k queries are deferred to Section 5. However, one undesirable problem with the PT-k query is that the number of tuples returned may differ a lot over different databases even when using the same threshold value. The user must set the threshold carefully to make the result set contain k tuples. Therefore, we study a slight variation of it, namely, the *probabilistic k top-k*, or simply Pk-top$k$. In the Pk-top$k$ query, we do not set a threshold, but return k tuples with the highest aggregate probabilities being one of the top-k ranked tuples in a random possible world. Formally, the Pk-top$k$ query is defined as follows:

Definition 1.1. *Probabilistic k top-k query (Pk-top$k$)*: Let D denote an uncertain database, $\mathcal{PW}$ the possible world space for D. Let $PW(t_i) \subseteq \mathcal{PW}$ denote the set of possible worlds containing t_i as one of its top-k ranked tuples. A Pk-top$k$ query returns a set of k tuples $T = \{t^1, \cdots, t^k\}$, satisfying $\sum_{pw \in PW(t^i)} \Pr[pw] \geq \sum_{pw \in PW(t^j)} \Pr[pw]$, for any $t_i \in T$ and $t_j \notin T$.

The Pk-top$k$ query returns the k most probable tuples of being the top-k among all. For example, in the uncertain dataset of Table 1 and with k = 2, the answer is $\{5, 6\}$, as they are the two with the highest probabilities of being among the top-k, with probabilities 0.64 and 0.5, respectively.

All the existing approaches for processing top-k queries [20, 21, 28, 30, 33] are designed for static uncertain datasets, and are incapable of handling streaming data. Directly applying the previous solutions on sliding windows would require storing all the tuples within the window, which is quite memory consuming. One major challenge is that the number of possible worlds that change as the window slides for one timestamp is huge. Assuming that there are W tuples in the window, then the number of possible worlds is 2^W. When a new tuple arrives and an old tuple fades out, $3/4$ of the 2^W possible worlds will change.

Consider a continuous Pk-top$k$ query with $k = 2$ over a sliding window of size $W = 3$, evaluated on the example in Table 1. Initially the first three tuples arrive (speed readings are 5, 6, and 8). The Pk-top$k$ answer is $\{6, 5\}$. Then, when the fourth tuple (speed reading 2) arrives, the first speed reading 5 expires, and the possible worlds are built over $\{6, 8, 2\}$. Now the top-2 answer becomes $\{6, 8\}$. It is important to note that the highest speed reading, 8, is not included in the top-k answer in the first sliding window, but is included in the top-k answer in the second sliding window. This example shows that in the sliding-window setting, in order to lower the memory requirement, we need a delicate and efficient mechanism to decide which tuples shall and shall not be kept for answering the query as the window advances through time.

Our contributions. In this paper, we design a unified framework for processing continuous top-k queries in a sliding window over uncertain data streams. All of the previously proposed top-k definitions can be plugged into our framework. Our framework is composed of several space- and time-efficient synopses with provable bounds. As depicted in Section 3, while it is relatively easy to handle arriving tuples, it is much difficult to cope with tuples expir-

ing. We need a carefully designed synopsis storing the minimum amount of information while sufficient for answering the query continuously at all times, which can also be maintained efficiently.

After formally defining the problem (Section 2), we first show how the previous techniques can be adapted to be self-maintainable with respect to insertions (Section 3). This automatically gives us a solution if tuples only arrive but never expire, which corresponds to the case of unbounded streams (or the so-called *landmark windows*). However, handling deletions is inherently much more difficult than insertions. In fact, if deletions are arbitrary, there is no better way than keeping everything in the sliding window, since each tuple would have a chance of being in the query results. Thus, in order to lower the space complexity, we need to exploit the important property of a sliding window that tuples arriving first will also expire first. In Section 4, we propose a series of synopses, each one building upon the previous one, that progressively improve the space and time complexities. These synopses are the results of a novel combination of several techniques including data compression, buffering, and ideas from exponential histograms [15]. We also analytically prove their space and time bounds, showing that although our synopses use much less space than the window size, we can still match the best running time one can hope for, even if linear space is allowed. Our analytical results are then further supported by an experimental report in Section 6, where we observe order-of-magnitude improvements over the previous solutions adapted to the sliding window model. In Section 5, we further discuss how the other top-k definitions can be plugged into our framework. Finally, we review the related work in Section 7 before concluding the paper.

2. PROBLEM STATEMENT

Let T be an uncertain stream containing a sequence of tuples, $t^1, t^2, \ldots, t^N$, where the superscripts denote the timestamps of the tuples. Let f be a ranking function. We use $t^i \prec_f t^j$ if $f(t^i) > f(t^j)$, and we say t^i's rank is higher than t^j's. In a similar fashion, $t^i \succ_f t^j$ means t^i's rank is lower than t^j's. Without loss of generality, we assume that the ranks of all tuples are unique. The membership probability of tuple t is denoted as $p(t)$.

A sliding window starting at position i and ending at j is denoted as $S[i, j]$, i.e., $S[i, j] = (t^i, t^{i+1}, \cdots, t^j)$, for $i \leq j$. The size of the sliding window is $\mathsf{wsize}(S[i, j]) = j - i + 1$. For the sliding window $S[i, j]$, $\mathcal{PW}(S[i, j])$ denotes its possible world space $\mathcal{PW}(S[i, j]) = \{PW_1, PW_2, \cdots\}$, where PW_j is a possible world that is a subset of tuples in $S[i, j]$. The probability of such a possible world PW is given as $\Pr(PW) = \Pi_{t \in PW} p(t) \times \Pi_{t \notin PW}(1 - p(t))$.

Problem statement. Given an uncertain data stream $T = (t^1, t^2, \ldots, t^N)$, and a sliding window size W, the goal is to answer the top-k query for every sliding window $S[i - W + 1, i]$ as i goes from W to N. For now we will use tuple-based windows, where at time i, t^i arrives while t^{i-W} expires. But all our algorithms can be easily extended to time-based windows. We will mostly focus on the Pk-top$k$ query, but will also discuss extensions to the other queries in Section 5. As with all streaming algorithms [3], memory consumption is the most important measure; but at the same time, we would like the processing time per tuple to be as low as possible.

3. COMPACT SET

This section first defines the *compact set*, a basic concept in all our synopses. It turns out if there are only insertions, one single compact set is sufficient for maintaining the top-k answers. However, we need multiple compact sets combined together to cope with expiring tuples.

Suppose the tuples in an uncertain dataset D are $t_1 \prec_f \cdots \prec_f t_n$. Denote by D_i the subset of D containing the first i tuples in D, $D_i = \{t_1, \cdots, t_i\}$. For $0 \leq j \leq i \leq n$, let $r_{i,j}$ be the probability that a randomly generated world from D_i has exactly j tuples. It is clear that the probability that t_i ranks the j-th in a randomly generated world from D is $p(t_i) \cdot r_{i-1,j-1}$.

Definition 3.1. The *compact set* $C(D)$ for the Pk-top$k$ query on an uncertain data set D is the smallest subset of D that satisfies the following conditions. (1) $\forall t' \in C(D)$ and $t'' \in D - C(D)$, $t' \prec_f t''$. (2) Let $d = |C(D)|$, t_d the tuple with the lowest rank in $C(D)$. There are k tuples in $C(D)$, and each such tuple t_α has

$$p(t_\alpha) \sum_{1 \leq l \leq k} r_{\alpha-1,l-1} \geq \sum_{1 \leq l \leq k} r_{d,l-1}. \quad (1)$$

Note that D may not always have a compact set, that is, even if we put all tuples into $C(D)$, (1) still cannot be satisfied. When there exists a $C(D)$ such that (1) holds, we say that D *admits* a compact set.

It is not difficult to obtain the following recursion [20, 33]:

$$r_{i,j} = \begin{cases} p(t_i) r_{i-1,j-1} + (1 - p(t_i)) r_{i-1,j}, & i \geq j \geq 0; \\ 1, & i = j = 0; \\ 0, & \text{else.} \end{cases} \quad (2)$$

Thus we can use dynamic programming to compute all the entries in the array r, as well as $C(D)$, in time $O(kd)$. We first show that if D has a compact set $C(D)$, then we do not need to look at tuples not in $C(D)$ in order to answer a Pk-top$k$ query.

Theorem 3.1. *The compact set $C(D)$ is sufficient for answering a Pk-top$k$ query on D.*

PROOF. Consider any tuple t_i, $i > d$. Let ξ_s be the probability that exactly s tuples from $\{t_{d+1}, \cdots, t_{i-1}\}$ appear. The probability that t_i's rank is j is $p(t_i)(\sum_{l=1}^{j} r_{d,l-1}\xi_{j-l})$. Then, the probability of t_i being ranked at any position between 1 and k is

$$p(t_i) \sum_{j=1}^{k} \left(\sum_{l=1}^{j} r_{d,l-1}\xi_{j-l} \right) = p(t_i) \sum_{l=1}^{k} \left(r_{d,l-1} \sum_{j=0}^{k-l} \xi_j \right)$$

$$\leq \quad p(t_i) \sum_{j=1}^{k} r_{d,j-1} \leq \sum_{j=1}^{k} r_{d,j-1},$$

where the first "$\leq$" is because $\sum_{j=0}^{k-1} \xi_j \leq 1$. □

Thus if D has a compact set $C(D)$, then we only need to run the dynamic program on $C(D)$ to compute the Pk-top$k$ results in time $O(kd)$. This algorithm is actually similar to that in [20, 21, 33], which also show that except for some pathological cases, the compact set almost always exists and much smaller than the whole data set. So answering a top-k query is usually quite efficient, and we do not need to look at the entire dataset at all, assuming of course the tuples are already sorted in rank order.

Example 3.1. Consider a Pk-top$k$ query over the dataset in Table 1, $k = 2$. After sorting, $t_1 = 8$, $t_2 = 6$, $t_3 = 5$, and $t_4 = 2$. Applying (2), we calculate the array r using dynamic programming: $r_{0,0} = 1$, $r_{0,1} = 0$, $r_{1,0} = 0.6$, $r_{1,1} = 0.4$, $r_{2,0} = 0.3$, $r_{2,1} = 0.5$, $r_{3,0} = 0.06$, $r_{3,1} = 0.34$. Now, we find that when $d < 3$, there does not exist any tuple t_α ($\alpha \leq d$) satisfying (1). When $d = 3$, all of three tuples (t_1, t_2, and t_3) are valid t_α. So, the compact set for that dataset is $\{t_1, t_2, t_3\}$. □

However, [20, 21, 33] only considered the static case. It is not clear at all whether this compact set can be self-maintained as tuples are inserted into D. It turns out that to answer this question, a much more careful analysis is required.

Self-maintenance of the compact set. We first need to study some important characteristics of the array r. We also study the change ratio $q_{i,j}$ of adjacent entries for any tuple t_i, namely $q_{i,j} = \frac{r_{i,j+1}}{r_{i,j}}$. Specifically, we can prove the following properties.

Lemma 3.1. *The value of $q_{i,j}$ is monotonically decreasing for any tuple t_i, i.e., $r_{i,j}^2 \geq r_{i,j-1} \cdot r_{i,j+1}$, for $j \geq 1$.*

PROOF. When $i = 1$, $r_{i,0} = 1 - p(t_1)$, $r_{i,1} = p(t_1)$, and for any $j > 1$, $r_{i,j} = 0$. So, when $i = 1$, the lemma holds.

Assuming the lemma is true for i, we consider the case of $i + 1$. If $j + 1 > i$, it is trivial because $r_{i,j+1} = 0$. So, we only need to analyze the situation where $1 \leq j < i$. By (2),

$$\begin{aligned}
\Delta &= r_{i+1,j}^2 - r_{i+1,j-1} r_{i+1,j+1} \\
&= (p(t_{i+1}) r_{i,j-1} + (1 - p(t_{i+1})) r_{i,j})^2 \\
&\quad -(p(t_{i+1}) r_{i,j-2} + (1 - p(t_{i+1})) r_{i,j-1}) \cdot \\
&\quad\ (p(t_{i+1}) r_{i,j} + (1 - p(t_{i+1})) r_{i,j+1}) \\
&= (p(t_{i+1}))^2 (r_{i,j-1}^2 - r_{i,j-2} r_{i,j}) \\
&\quad +(1 - p(t_{i+1}))^2 (r_{i,j}^2 - r_{i,j-1} r_{i,j+1}) \\
&\quad +p(t_{i+1})(1 - p(t_{i+1}))(r_{i,j-1} r_{i,j} - r_{i,j-2} r_{i,j+1}) \\
&\geq p(t_{i+1})(1 - p(t_{i+1}))(r_{i,j-1} r_{i,j} - \frac{r_{i,j-1}^2}{r_{i,j}} \frac{r_{i,j}^2}{r_{i,j-1}}) \quad (3) \\
&= 0.
\end{aligned}$$

Note that we assumed $r_{i,j-1} > 0$ and $r_{i,j} > 0$ in (3). Otherwise, $r_{i,j-2} = 0$, we still have $\Delta \geq 0$. □

Lemma 3.2. *For any two tuples t_i, t_j satisfying $t_i \prec_f t_j$, we have $q_{i,l} \leq q_{j,l}$, where $l \geq 0$, i.e., $r_{i,l+1} r_{j,l} \leq r_{i,l} r_{j,l+1}$.*

PROOF. First, consider the case $j = i + 1$. We have

$$\begin{aligned}
q_{i,l} - q_{i+1,l} &= \frac{r_{i,l+1}}{r_{i,l}} - \frac{r_{i+1,l+1}}{r_{i+1,l}} \\
&= \frac{1}{r_{i,l} r_{i+1,l}} (r_{i,l+1} r_{i+1,l} - r_{i,l} r_{i+1,l+1}) \\
&= \frac{p(t_{i+1})}{r_{i,l} r_{i+1,l+1}} (r_{i,l+1} r_{i,l-1} - r_{i,l}^2) \leq 0 \quad \text{(Lemma 3.1)}
\end{aligned}$$

Repeating the same step iteratively proves the lemma for any $j > i$. □

Lemma 3.3. *For any tuple t_i, the series $r_{i,j}$ is unimodal, i.e., there exists some m such that $r_{i,j}$ is monotonically increasing when $j < m$ while monotonically decreasing when $j > m$.*

PROOF. According to Lemma 3.1, the value of $q_{i,j}$ decreases monotonically. Let m be the maximum such that $q_{i,m} \leq 1$, then it is not difficult to verify that m meets the requirement in the lemma. □

Lemma 3.4. *For any tuples t_i, t_j, $t_i \prec_f t_j$. The peak point of the corresponding series (in Lemma 3.3) for t_i is no later than t_j.*

PROOF. Follows from Lemma 3.2. □

Theorem 3.2. *Let $C(D)$ be the compact set of D, let t_d be the lowest-rank tuple in $C(D)$, and let t_{new} be a new tuple to be inserted into D. Then $C(D \cup \{t_{new}\}) = C(D)$ if $t_d \prec_f t_{new}$, and $C(D \cup \{t_{new}\}) \subseteq C(D) \cup \{t_{new}\}$ if $t_{new} \prec_f t_d$.*

PROOF. Let r' be the array for $C(D) \cup \{t_{new}\}$. Let us consider the following cases in turn.

Case 1: $t_{new} \succ_f t_d$. Then $r_{i,j} = r'_{i,j}$ for all $1 \leq i \leq d$, so $r_{\alpha-1,l-1}$ and $r_{d,l-1}$ remain unchanged for $1 \leq l \leq k$. Thus, $C(D \cup \{t_{new}\}) = C(D)$.

Case 2: For all the t_α that meets (1), $t_d \succ_f t_{new} \succ_f t_\alpha$. We will show that (1) still holds on r'. For $1 \leq l \leq k$, we have

$$r'_{d,l-1} = p(t_{new}) r_{d,l-2} + (1 - p(t_{new})) r_{d,l-1}.$$

Summing over all l,

$$\sum_{l=1}^{k} r'_{d,l-1} = \sum_{l=1}^{k} r_{d,l-1} - p(t_{new}) r_{d,k-1}, \quad (4)$$

namely, the RHS of (1) is reduced while its LHS stays the same. So (1) still holds on r', hence $C(D \cup \{t_{new}\}) \subseteq C(D) \cup \{t_{new}\}$.

Case 3: There exists one or more t_α that meet (1) such that $t_\alpha \succ_f t_{new}$. Now both the LHS and RHS of (1) change, so we need to be more careful.

First, for any such t_α, we have

$$r'_{\alpha-1,l-1} = p(t_{new}) r_{\alpha-1,l-2} + (1 - p(t_{new})) r_{\alpha-1,l-1}.$$

Summing over all l,

$$\sum_{l=1}^{k} r'_{\alpha-1,l-1} = \sum_{l=1}^{k} r_{\alpha-1,l-1} - p(t_{new}) r_{\alpha-1,k-1}.$$

So the LHS of (1) decreases by a fraction of

$$1 - \frac{p(t_\alpha) \sum_{l=1}^{k} r'_{\alpha-1,l-1}}{p(t_\alpha) \sum_{l=1}^{k} r_{\alpha-1,l-1}} = p(t_{new}) \frac{r_{\alpha-1,k-1}}{\sum_{l=1}^{k} r_{\alpha-1,l-1}}.$$

Similarly, by (4), the RHS of (1) decreases by a fraction of

$$p(t_{new}) \frac{r_{d,k-1}}{\sum_{l=1}^{k} r_{d,l-1}}.$$

Next we show that

$$\frac{r_{d,k-1}}{\sum_{l=1}^{k} r_{d,l-1}} \geq \frac{r_{\alpha-1,k-1}}{\sum_{l=1}^{k} r_{\alpha-1,l-1}}, \quad (5)$$

thus establishing the fact that (1) still holds on r'.

We prove (5) by induction. For the base case $k = 2$, by Lemma 3.2, we have

$$\begin{aligned}
\frac{r_{d,1}}{r_{d,0} + r_{d,1}} &= 1 - \frac{r_{d,0}}{r_{d,0} + r_{d,1}} \geq 1 - \frac{r_{d,0}}{r_{d,0} + \frac{r_{\alpha-1,1} r_{d,0}}{r_{\alpha-1,0}}} \\
&= \frac{r_{\alpha-1,1}}{r_{\alpha-1,0} + r_{\alpha-1,1}}.
\end{aligned}$$

Next we consider $k + 1$ assuming (5) is true for k. Again by Lemma 3.2, we have

$$\begin{aligned}
\frac{r_{d,k}}{\sum_{l=1}^{k+1} r_{d,l-1}} &\geq \frac{\frac{r_{\alpha-1,k} r_{d,k-1}}{r_{\alpha-1,k-1}}}{\sum_{l=1}^{k} r_{\alpha-1,l-1} \frac{r_{d,k-1}}{r_{\alpha-1,k-1}} + \frac{r_{\alpha-1,k} r_{d,k-1}}{r_{\alpha-1,k-1}}} \\
&= \frac{r_{\alpha-1,k}}{\sum_{l=1}^{k+1} r_{\alpha-1,l-1}}.
\end{aligned}$$

So (5) holds for all k, and the theorem is proved. □

Theorem 3.2 gives us a very simple algorithm for maintaining the compact set if tuples are only inserted into D but never deleted. For an incoming tuple t_{new}, we first check if $t_{new} \prec_f t_d$. If so then we recompute the array r on $C(D) \cup \{t_{new}\}$, which gives us the updated compact set $C(D \cup \{t_{new}\})$ and also the updated top-k results. Otherwise we simply discard t_{new} knowing that it will not

affect the query results. However, the presence of expiring tuples make the problem much more difficult, since if a tuple in $C(D)$ expires, this whole compact set is useless and we need to compute a new compact set from D. Thus simply using one compact set for a sliding window implies that we cannot discard any tuple in the window until it expires, using memory $\Omega(W)$. In the next section, we present our sliding-window synopses, which combines multiple compacts sets together, so that we can safely discard most tuples in the window while still being able to maintain the up-to-date query results at any time as the window slides through time.

4. SYNOPSES FOR SLIDING WINDOWS

The previous section shows that the compact set is self-maintainable under insertions. However, if a tuple in the compact set expires, then there is no way to reconstruct it without maintaining tuples outside the compact set. Then the question is, how many other tuples do we need to keep? This section will focus on answering this question.

First of all, notice that in the worst-case scenario, for example when the tuples always arrive with decreasing ranks and decreasing probabilities, any tuple will be in the top-k result at some point in time as the window slides. In this case, any synopsis has to remember everything in the window in order to avoid incorrect query results. So it is hopeless to design a synopsis with a sublinear worst-case space bound. Therefore, we will assume that the tuples arrive in a random order. This *random-order stream* model has received much attention lately from the stream algorithms community [18, 6, 5], mainly because the worst-case bounds for many streaming problems are simply too pessimistic and thus meaningless. The random-order stream model has been argued to be a reasonable approximation of real-world data streams while often allowing for much better expected bounds. This model is an ideal choice for the study of our problem since as shown above, in the worst case, there is really nothing better one can do than the naive approach, which simply keeps all tuples in the sliding window.

Before presenting our solutions, we first analyze the direct adaptation of the existing technique to the sliding window setting, which we refer to as the *Base Synopsis*, or the *BS*. To make the analytical comparison with our synopses easier, we use H to denote the maximum size of the compact sets that are maintained in the synopsis. As argued in [33, 20], although in the worst case, $H = W$, but on most datasets, $H \ll W$. As discussed in the previous section, BS needs to keep all the W tuples in the window (in the rank order) and its compact set C. The array r takes $O(kH)$ space, thus the total space of BS is $O(W + kH)$, which is effectively $O(W)$ since $H \ll W$. When the window slides, if either the expiring tuple is in C, or the incoming tuple's rank is higher than the lowest ranked tuple in C, then we recompute C from all the tuples in the window. Since C keeps the highest-ranked tuples in the window, either event happens with probability $O(H/W)$, so the expected cost of maintaining C is $O(kH^2/W)$. Maintaining the tuples in the rank order takes $O(\log W)$ time per tuple. Thus we have the following.

Lemma 4.1. *BS requires $O(W + kH)$ space and spends expected $O(kH^2/W + \log W)$ time to process each tuple.*

In the following subsections we present our sliding-window synopses. Each of them builds upon the previous one with new ideas, progressively improving either the space complexity or the processing time. Our final synopsis requires $O(H(k + \log W))$ space and has a processing time of $O(kH^2/W + \log W)$. So it matches the processing time of BS while having a much lower space complexity. To appreciate this result, the reader is reminded that most streaming algorithms, e.g., most sketches [3, 25], require higher running times than the naive approach in order to achieve low space complexity.

Algorithm 1 MaintainCSQ

```
1: Tuple set D = ∅; compact set queue Ψ = ∅;
2: for each arriving tuple t
3:    insert t into D;
4:    if (successfully create a compact set C(D) for D)
5:       append C(D) to Ψ;
6:       remove tuples in D older than t'' (including t''), where
         t'' is the oldest tuple in C(D);
7:    for (each compact set C(S_i) ∈ Ψ from new to old)
8:       if (t ≺_f lowest ranked tuple in C(S_i))
9:          update C(S_i) := C(C(S_i) ∪ {t});
10:         if (C(S_i) = the previous compact set in Ψ)
11:            remove C(S_i) from Ψ;
12:      else
13:         break;
14:   if (the expiring tuple ∈ C(S_W))
15:      remove C(S_W) from Ψ;
16:      C(S_W) := first compact set in Ψ;
17:      compute the array r on the new C(S_W);
```

4.1 Compact Set Queue

Our first synopsis, called the *Compact Set Queue* (*CSQ*) is the simplest of all but forms the basis of the more advanced synopses. Let S_i denote the set of the last i tuples in the sliding window. In the CSQ, we simply keep all the distinct compact sets $C(S_i)$ for all $i = 1, \ldots, W$. We only keep the array r for $C(S_W)$ from which we can extract the top-k results. Since we have the compact set for each S_i, when a tuple in $C(S_W)$ expires, we can move $C(S_{W-1})$ forward to become the new $C(S_W)$.

Algorithm 1 describes the detailed algorithm to maintain the CSQ. We maintain a queue Ψ of all the distinct compact sets. The tuple set D temporarily keeps the newest tuples. Initially D does not admit a compact set. As tuples arrive at D, D will have a valid compact set at some point. When this happens, we create $C(D)$, and append it to Ψ. Tuples in D but older than the oldest tuple in $C(D)$ (including the oldest tuple in $C(D)$) are removed from D (lines 1–6). Note that after the removal, D does not admit a compact set anymore. Therefore, when D has collected enough new tuples, the new compact set it generates must be different from the existing ones in Ψ. Next we update all the compact sets in Ψ in turn according to Theorem 3.2, while removing duplicates (lines 7–13). Finally, we check if the expiring tuple exists in $C(S_W)$, if so we remove $C(S_W)$ from Ψ, and the next compact set in Ψ becomes the new $C(S_W)$ (lines 14–17).

Lemma 4.2. *The expected number of distinct compact sets in CSQ is $O(H \log W)$; the expected number of compact sets that need to be updated per tuple is $O(H)$.*

PROOF. Let X_i be the indicator variable such that $X_i = 1$ if $C(S_i)$ is different from $C(S_{i+1})$, and $X_i = 0$ otherwise. It is obvious that the expected number of distinct compact sets in Ψ is $\mathbf{E}[\sum_{i=1}^{W} X_i]$. The event $C(S_{i+1}) \neq C(S_i)$ happens only when the rank of the oldest tuple in S_{i+1} is higher than the lowest ranked tuple in $C(S_i)$. Because $C(S_i)$ contains the $\leq H$ highest ranked tuples in S_i, the occurring probability of this event is at most H/i. Hence,

$$\mathbf{E}[X_i] = \Pr[X_i = 1] \leq \max\{1, H/i\},$$

and

$$\mathbf{E}\left[\sum_{i=1}^{W} X_i\right] \le H + H\left(\frac{1}{H} + \cdots + \frac{1}{W}\right) = O(H \log W).$$

Now consider the arrival of a new tuple t. Let Y_i be the indicator variable such that $Y_i = 1$ iff $X_i = 1$ and t affects $C(S_{i+1})$. For the latter to happen, t must rank higher than the lowest ranked tuple in $C(S_{i+1})$, so $\Pr[Y_i = 1 | X_i = 1] \le \max\{1, H/(i+1)\}$. Hence, the expected number of compact sets affected by t is

$$\begin{aligned}\mathbf{E}\left[\sum_{i=1}^{W} Y_i\right] &= \sum_{i=1}^{W} \Pr[X_i = 1] \Pr[Y_i = 1 | X_i = 1] \\ &\le H + H^2\left(\frac{1}{H(H+1)} + \cdots + \frac{1}{W(W+1)}\right) \\ &\le H + H^2 \cdot \frac{1}{H} = O(H).\end{aligned}$$

□

Theorem 4.1. *CSQ requires $O(H^2 \log W)$ space and can be maintained in time $O(kH^2)$ per tuple.*

PROOF. Since each compact set has size $O(H)$, and the array r has size $O(kH)$, the space bound follows from Lemma 4.2. Each compact set can be updated in time $O(kH)$ and there are $O(H)$ of them that need to be updated, so the total time for the update is $O(kH^2)$. □

4.2 Compressed Compact Set Queue

Although CSQ only contains distinct compact sets, there is still a lot of redundancy as one tuple may appear in multiple compact sets. In the *Compressed Compact Set Queue (CCSQ)*, we try to eliminate this redundancy by storing only the difference between two adjacent compact sets $C(S_i)$ and $C(S_{i-1})$. More precisely, if $C(S_i) \ne C(S_{i-1})$, we keep both $\Delta_i^+ = C(S_i) - C(S_{i-1})$ and $\Delta_i^- = C(S_{i-1}) - C(S_i)$. Now we can discard all the $C(S_i)$ in the queue Ψ except the newest one.

We need to bound the total size of these differences. First, since S_i has only one more tuple than S_{i-1}, by Theorem 3.2, it is clear that $|\Delta_i^+| \le 1$. By Lemma 4.2, the total number of nonempty Δ_i^+ is $O(H \log W)$, so we have $\sum_{i=1}^{W} |\Delta_i^+| = O(H \log W)$. To bound the total size of all the Δ_i^-, we need the following property.

Lemma 4.3. *If tuple t appears both in $C(S_i)$ and $C(S_j)$, $i < j$, then it appears in all compact sets between $C(S_i)$ and $C(S_j)$, i.e., $t \in C(S_l)$ for all $i \le l \le j$.*

PROOF. Because $S_j \supset S_l$, by repeatedly applying Theorem 3.2, we have $C(S_j) \subseteq C(S_l) \cup (S_j - S_l)$, i.e., any tuple in $C(S_j)$ is either from $C(S_l)$ or from $S_j - S_l$. For any $t \in C(S_i) \cap C(S_j)$, since $t \in S_i$ and $S_i \cap (S_j - S_l) = \emptyset$, we must have $t \in C(S_l)$. □

Thus, as we go from $C(S_1)$ to $C(S_W)$, once a tuple disappears, it will never appear again. So we have

$$\sum_{i=1}^{W} |\Delta_i^-| \le H + \sum_{i=1}^{W} |\Delta_i^+| = O(H \log W).$$

By this compression technique, we have reduced the space complexity of CSQ by roughly an $O(H)$ factor.

Theorem 4.2. *CCSQ requires $O(H(k + \log W))$ space and can be maintained in time $O(kH^2)$ per tuple.*

PROOF. As argued above, storing all the compact sets with compression requires $O(H \log W)$ space. We also need the array r, which takes $O(kH)$ space.

To see that the processing time remains unchanged, just note that Lemma 4.2 still holds, and we can restore each $C(S_i)$ in Ψ by making a pass over Δ_i^+ and Δ_i^-, update it, and compute the new Δ_i^+ and Δ_i^-, all in time $O(kH^2)$. □

4.3 Segmental Compact Set Queue

With CCSQ, we have lowered the space complexity of the synopsis to almost minimal: we only need one array r and keep $O(H \log W)$ tuples, as opposed to BS which stores all the W tuples. However, the maintenance cost of CCSQ is still very high. In the next two advanced synopses, we try to improve the processing time while maintaining the low space complexity.

We notice that the high computation complexity is due to the fact that $O(H)$ compact sets need to be updated per incoming tuple. However, only the oldest compact set $C(S_W)$ is needed to extract the top-k query results; all the other compact sets simply act as a continuous "supply" for $C(S_W)$ when it expires. For these compact sets, we actually do not need to maintain them exactly. As long as we have a super set for each of them, which can be maintained much more efficiently, then we can still reconstruct it exactly when it becomes the oldest compact set in the queue. But on the other hand, we do not want these super sets to be too large to violate the space constraint, so we need a carefully designed mechanism to balance space and time. With this intuition, we introduce our next synopsis, the *Segmental Compact Set Queue (SCSQ)*.

In SCSQ, we only maintain a small number of distinct compact sets $C(S_{\ell_1}), \cdots, C(S_{\ell_n})$, for $1 \le \ell_1 < \cdots < \ell_n \le W$. For each i, we also maintain Λ_{ℓ_i}, a set of tuples in $S_{\ell_{i+1}} - S_{\ell_i}$ (define $\ell_{n+1} = W$) such that $C(S_j) \subseteq S_{\ell_i} \cup \Lambda_{\ell_i}$ for all $\ell_i \le j < \ell_{i+1}$. Note that any tuple in Λ_{ℓ_i} must rank higher than the lowest ranked tuple in $C(S_{\ell_i})$. Finally, we always keep $C(S_W)$ and its associated array r, from which we extract the top-k results.

We maintain the following invariants in SCSQ throughout time[1]:

$$|\Lambda_{\ell_i}| \le H, \qquad \text{for } i = 1, \ldots, n; \tag{6}$$

$$|\Lambda_{\ell_i}| + |\Lambda_{\ell_{i+1}}| \ge H, \qquad \text{for } i = 1, \ldots, n-1. \tag{7}$$

The correctness of SCSQ follows from its definition and Theorem 3.2: Whenever $C(S_W)$ expires, since the new $C(S_W)$ is a subset of $C(S_{\ell_n}) \cup \Lambda_{\ell_n}$, we can rebuild it in time $O(kH)$.

Whenever invariant (7) is violated, we do a merge by setting $\Lambda_{\ell_i} := \Lambda_{\ell_i} \cup \Lambda_{\ell_{i+1}} \cup \{t'\}$, where t' is the oldest tuple in $C(S_{\ell_i+1})$, and then removing $C(S_{\ell_{i+1}}), \Lambda_{\ell_{i+1}}$. It is not difficult to verify that $\Lambda_{\ell_i} \cup C(S_{\ell_i})$ now contains all the tuples needed to cover any $C(S_j)$ for $\ell_i \le j < \ell_{i+2}$, and both invariants (6) and (7) are restored.

The procedure to maintain the SCSQ is actually very similar to that of CSQ, the only difference is now we only update $C(S_W)$ and $C(S_{\ell_i})$ for each $i = 1, \ldots, n$. Next, if $C(S_{\ell_i})$ has changed, tuples in Λ_{ℓ_i} are simply removed if their ranks are lower than the lowest ranked tuple in $C(S_{\ell_i})$. Whenever invariant (7) is violated, we do a merge as described above. Finally, if $C(S_W)$ expires, we compute a new $C(S_W)$ from $C(S_{\ell_n}) \cup \Lambda_{\ell_n}$.

Example 4.1. Figure 2 shows how the SCSQ evolves over time. For illustration purposes we assume $H = 3$ and all compact sets have exactly the 3 highest ranked tuples. The queue contains two

[1]Note that H is not fixed in advance and may change over time. So we update and use a new H whenever the maximum size of the compact sets currently maintained in the synopsis changes by a factor of 2. This does not affect the asymptotic bounds of our algorithms.

compact sets at time 8: $C(S_3) = \{5, 6, 1\}$, $\Lambda_3 = \emptyset$; $C(S_4) = \{3, 5, 6\}$, $\Lambda_4 = \{8, 7, 9\}$. When tuple t_9 arrives, the existing two compact sets are shifted and updated as $C(S_4) = \{5, 6, 4\}$, $C(S_5) = \{5, 6, 4\}$. At the same time, Λ_3 and Λ_4 are also shifted (but unchanged) to be Λ_4 and Λ_5. Since $C(S_4)$ and $C(S_5)$ are now the same, we delete $C(S_5)$, and set $\Lambda_4 = \Lambda_5$. A new compact set is created $C(S_3) = \{5, 6, 1\}$ and $\Lambda_3 = \emptyset$. Next, we remove the expiring tuple 8 from Λ_4. Since $|\Lambda_3|+|\Lambda_4| = 2 < H$, we do a merge, removing $C(S_4)$ while updating $\Lambda_3 := \Lambda_3 \cup \Lambda_4 \cup \{5\} = \{7, 9, 5\}$. The final status is shown in Figure 2(b). □

The following result is crucial in bounding the size and processing time of SCSQ.

Lemma 4.4. *SCSQ maintains expected $O(\log W)$ compact sets.*

PROOF. Under the random-order stream model, all of $\ell_1, \ldots, \ell_n$, as well as n, are random variables. Below we show that $\mathbf{E}[n] = O(\log W)$.

Consider the stochastic process consisting of the sequence of random variables $\ell_1, \ell_3, \ell_5, \ldots$. We say that it is a good event if $\ell_{2i+1} \geq 2\ell_{2i-1}$, and a bad event otherwise. It is clear that the sequence will terminate before we have $\log W$ good events. We construct a sequence of indicator variables $X_1, X_2, \ldots$, where $X_i = 1$ iff the i-th event is good, and let $Y_m = X_1 + \cdots + X_m$. Then $\mathbf{E}[n] \leq 2 \cdot \mathbf{E}[\arg\min_m \{Y_m = \log W\}]$.

Now we focus on bounding $\mathbf{E}[Y]$. Consider the bad event $\ell_{2i+1} < 2\ell_{2i-1}$. If this happens, due to invariant (7), there must be more than H tuples in $S_{2\ell_{2i-1}} - S_{\ell_{2i-1}}$ that rank higher than the H-th ranked tuple in $S_{\ell_{2i-1}}$. If so, among the top-$(2H)$ ranked tuples in $S_{2\ell_{2i-1}}$, more than half of them must be in the older half $S_{2\ell_{2i-1}} - S_{\ell_{2i-1}}$. This occurs with probability less than 1/2. Therefore, for any i, the probability that the i-th event is bad is less than 1/2, or equivalently $\Pr[X_i = 1] > 1/2$.

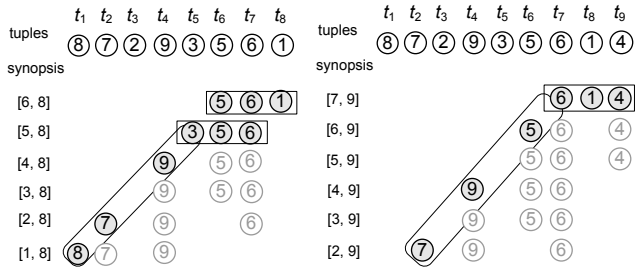
Although $X_1, X_2, \ldots$ are not necessarily independent, the argument above holds for each X_i regardless of the values of $X_j, j \neq i$. Therefore Y_m is *stochastically greater* than a binomial random variable $Z_m \sim \text{binomial}(m, 1/2)$: $Y_m \geq_{st} Z_m$. The expectation $\mathbf{E}[\arg\min_m \{Y_m = \log W\}]$ can be written as

$$
\begin{aligned}
& \sum_{i \geq 1} \Pr[\arg\min_m \{Y_m = \log W\} \geq i] \\
= & \sum_{i \geq 1} \Pr[Y_{i-1} < \log W] \\
\leq & \; 4 \log W + 1 + \sum_{i \geq 4 \log W} \Pr[Y_i < \log W] \\
\leq & \; O(\log W) + \sum_{i \geq 4 \log W} \Pr[Z_i < \log W] \quad (Y_m \geq_{st} Z_m) \\
\leq & \; O(\log W) + \sum_{i \geq 4 \log W} e^{-(i/2 - \log W)^2 / i} \quad \text{(Chernoff bound)} \\
\leq & \; O(\log W) + \sum_{i \geq 4 \log W} e^{-i/16} = O(\log W),
\end{aligned}
$$

hence the proof. □

Theorem 4.3. *SCSQ requires $O(H(k + \log W))$ space and processes each tuple in time $O(kH \log W)$.*

PROOF. Since the array r has size $O(kH)$ and each compact set has size $O(H)$, the space bound then follows from Lemma 4.4. Updating all the compact sets takes $O(kH \log W)$ time. Updating all the Λ_{ℓ_i} and doing the necessary merges take time $O(H \log W)$, hence the time is bounded. □



(a) At time 8 (b) At time 9

Figure 2: Maintaining the SCSQ.

4.4 SCSQ with Buffering

SCSQ makes an $O(H)$-factor improvement over the previous synopses in terms of processing time, but there is still room for improvement. With our final synopsis, *SCSQ-Buffer*, we make another significant improvement by augmenting SCSQ with a buffering technique, reducing the processing time to minimum.

The basic intuition here is that since only $C(S_W)$ is useful for the query, we update only this compact set every time the window slides. For the rest of the compact sets, we update them in batches. More precisely, we keep a buffer B of size kH for the latest tuples[2]. (We assume $W > kH$; otherwise we just switch to BS.) When the buffer is full, we empty it and make necessary changes to the synopsis. The detailed algorithm is shown in Algorithm 2.

Algorithm 2 BatchUpdate

```
1:  let B be a buffer with size kH;
2:  for (each arriving tuple t)
3:      insert t into B;
4:      if (B is full)
5:          find the smallest i such that B_i admits a compact set;
6:          starting from i, build SCSQ on B;
7:          update the existing SCSQ;
8:          B = ∅;
9:      if (C(S_W) is affected)
10:         update C(S_W);
11:     remove expired compact sets in SCSQ;
```

First, we need to build new compacts sets and the relevant Λ_ℓ's for the tuples in B. Let B_i be the set of i latest tuples in B. To do so, we first do a binary search to find the smallest i such that B_i admits a compact set (line 5). Since checking each B_i takes $O(kH)$ time, the binary search takes $O(kH \log(kH))$ time. Then we build the first compact set. Next we scan the remaining tuples from new to old, putting into Λ_i those tuples ranking higher than the lowest ranked one in $C(B_i)$. When $|\Lambda_i| = H$ we stop, and restart the same process by building another compact set. By Lemma 4.4 we will build $O(\log(kH))$ new compact sets for B, spending $O(kH \log(kH))$ time in total (line 6).

Secondly, we update all the existing compact sets $C(S_{\ell_i})$ and the Λ_{ℓ_i} with all the tuples in B (line 7). Since there are $O(\log W)$ compact sets and updating each one takes $O(kH)$ time, the total cost is $O(kH \log W)$. Updating all the Λ_{ℓ_i} and making all the necessary merges take $O(H \log W)$ time. Therefore, the total cost for emptying a buffer of size kH is $O(kH \log(kH) + kH \log W) = O(kH \log W)$. So the amortized cost per tuple is only $O(\log W)$.

Finally, for each incoming tuple, we always update $C(S_W)$ if necessary (line 9–10). Similar to the case with BS, $C(S_W)$ is affected with probability $O(H/W)$, so the cost of maintaining $C(S_W)$

[2]We change the size of the buffer whenever H changes by a factor of 2. See also footnote 1.

	Space	Processing time
BS	$O(W+kH)$	$O(kH^2/W+\log W)$
CSQ	$O(H^2\log W)$	$O(kH^2)$
CCSQ	$O(H(k+\log W))$	$O(kH^2)$
SCSQ	$O(H(k+\log W))$	$O(kH\log W)$
SCSQ-Buffer	$O(H(k+\log W))$	$O(kH^2/W+\log W)$

Table 2: Asymptotic space and processing time bounds analysis

is $O(kH^2/W)$.

Theorem 4.4. *SCSQ-Buffer requires $O(H(k+\log W))$ space and has an amortized processing time of $O(kH^2/W+\log W)$.*

We summarize the space and time complexities of all five synopses we have presented so far in Table 2. Bearing in mind that $k < H \ll W$, we can see that SCSQ-Buffer has both the best space bound and the best processing time.

5. SUPPORTING OTHER TOP-K DEFINITIONS

As we have seen, our synopses are quite general in the sense that any other top-k query definition can be plugged into the framework if a compact set can be defined such that Theorem 3.1 (sufficiency) and Theorem 3.2 (self-maintenance with respect to insertions) both hold. This section briefly shows how to support the other three top-k definitions on uncertain data proposed in the literature in our framework. In fact, all the existing solutions read the tuples in the rank order, and stop as soon as the correctness of the results are guaranteed. Such an approach naturally yields a compact set which is also sufficient. So we only need to prove self-maintainability.

PT-k queries. Let the array r be defined as before.

Definition 5.1.([20]) The *compact set* $C(D)$ for the PT-k query with a threshold τ on an uncertain data set D is the smallest subset of D that satisfies the following conditions. (1) $\forall t' \in C(D)$ and $t'' \in D - C(D)$, $t' \prec_f t''$. (2) $\tau \geq \sum_{1\leq l\leq k} r_{d,l-1}$, where t_d is the lowest ranked tuple in $C(D)$.

Theorem 5.1. *The compact set defined for PT-k queries is self-maintainable with respect to insertions.*

PROOF. Let t_{new} be the new tuple to be inserted to D. If $t_{new} \succ_f t_d$, $r_{d,l-1}$ remains unchanged for $1 \leq l \leq k$, so $C(D)$ stays unchanged. Otherwise if $t_{new} \prec_f t_d$, then

$$\begin{aligned}\sum_{l=1}^{k} r'_{d,l-1} &= \sum_{l=1}^{k}(p(t_{new})r_{d,l-2}+(1-p(t_{new}))r_{d,l-1})\\ &= \sum_{l=0}^{k-1} p(t_{new})r_{d,l-1}+\sum_{l=1}^{k}(1-p(t_{new}))r_{d,l-1}\\ &= \sum_{l=1}^{k} r_{d,l-1}-p(t_{new})r_{d,k-1} \leq \sum_{l=1}^{k} r_{d,l-1} < \tau.\end{aligned}$$

So, any tuple not in $C(D)\cup\{t_{new}\}$ cannot be an answer. □

U-kRanks Queries. Let the array r be defined as before.

Definition 5.2.([33]) The *compact set* $C(D)$ for the U-kRanks query on an uncertain data set D is the smallest subset of D that satisfies the following conditions. (1) $\forall t' \in C(D)$ and $t'' \in D - C(D)$, $t' \prec_f t''$. (2) Let t_d be the lowest ranked tuple in $C(D)$, then

$$\max_{1\leq i\leq d} p(t_i)r_{i-1,j-1} \geq \max_{1\leq l\leq k} r_{d,l-1}, \quad \text{for } j=1,\ldots,k. \qquad (8)$$

As defined, condition (8) is unwieldy to prove self-maintainability. So we first convert it to an equivalent, but much simpler condition. Specifically, we replace (8) with the following:

$$p(t_\alpha)r_{\alpha-1,k-1} \geq \max_{1\leq l\leq k} r_{d,l-1}, \quad \text{for some } \alpha \leq d. \qquad (9)$$

Compared with (8), (9) is much easier to check because it only requires finding one tuple t_α for rank k, not for all the ranks. But as we show below, (9) also implies (8), hence equivalent with (8).

Lemma 5.1. *For any i, if $p(t_i)r_{i-1,k-1} \geq \max_{1\leq l\leq k} r_{d-1,l-1}$, then for any j, $1 \leq j \leq k$, we have*

$$p(t_i)r_{i-1,j-1} \geq \max_{1\leq l\leq j} r_{d-1,l-1}. \qquad (10)$$

PROOF. Let $\arg\max_{1\leq l\leq k} r_{d-1,l-1} = m$. According to Lemma 3.3, $r_{d-1,l-1}$ monotonically increases when $l \leq m$ and monotonically decreases when $l \geq m$. By Lemma 3.2, when $l \geq m$, we have

$$r_{i-1,l-1} \geq \frac{r_{d-1,l-1}}{r_{d-1,l}} \cdot r_{i-1,l} \geq r_{i-1,l}.$$

So for all $m \leq j \leq k$,

$$p(t_i)r_{i-1,j-1} \geq p(t_i)r_{i-1,k-1} \geq \max_{1\leq l\leq k} r_{d-1,l-1} = \max_{1\leq l\leq j} r_{d-1,l-1}.$$

Next consider the case $1 \leq j \leq m$. Note that in this case the RHS of (10) is $r_{d-1,j-1}$. We will prove (10) by induction for $j = m,\ldots,1$. The base case $j = m$ has already been proved above. Now suppose (10) holds for j, i.e., $p(t_i)r_{i-1,j-1} \geq r_{d-1,j-1}$, and we consider $j-1$. By Lemma 3.2,

$$p(t_i)r_{i-1,j-2} \geq p(t_i)\frac{r_{i-1,j-1}}{r_{d-1,j-1}}r_{d-1,j-2} \geq r_{d-1,j-2}.$$

So, (10) holds for all j, $1 \leq j \leq k$. □

Theorem 5.2. *The compact set defined for U-kRanks query is self-maintainable with respect to insertions.*

PROOF. We consider the following three cases.

Case 1: $t_{new} \succ_f t_d$. In this case, $r_{\alpha-1,k-1}$ and $r_{d,l-1}$ remain unchanged, so $C(D)$ stays unchanged.

Case 2: $t_d \succ_f t_{new} \succ_f t_\alpha$. In this case, $r_{d,l-1}$ changes to

$$\begin{aligned}r'_{d,l-1} &= p(t_{new})r_{d,l-2}+(1-p(t_{new}))r_{d,l-1}\\ &\leq \max\{r_{d,l-2},r_{d,l-1}\} \leq \max_{1\leq l\leq k} r_{d,l-1},\end{aligned}$$

while $r_{\alpha-1,l-1}$ is unchanged. So (9) still holds on $C(D)\cup\{t_{new}\}$.

Case 3: $t_\alpha \succ_f t_{new}$. Both $r_{\alpha-1,l-1}$ and $r_{d-1,l-1}$ change to $r'_{\alpha-1,l-1}$ and $r'_{d-1,l-1}$. We compute

$$\begin{aligned}\Delta &= p(t_\alpha)r'_{\alpha-1,k-1}-\max_{1\leq l\leq k} r'_{d,l-1}\\ &= p(t_\alpha)(p(t_{new})r_{\alpha-1,k-2}+(1-p(t_{new}))r_{\alpha-1,k-1})\\ &\quad -\max_{1\leq l\leq k}\{p(t_{new})r_{d,l-2}+(1-p(t_{new}))r_{d,l-1}\}\\ &\geq p(t_\alpha)(p(t_{new})r_{\alpha-1,k-2}+(1-p(t_{new}))r_{\alpha-1,-1})\\ &\quad -(p(t_{new})\max_{1\leq l\leq k-1} r_{d,l-1}+(1-p(t_{new}))\max_{1\leq l\leq k} r_{d,l-1})\\ &= p(t_{new})(p(t_\alpha)r_{\alpha-1,k-2}-\max_{1\leq l\leq k-1} r_{d,l-1})\\ &\quad +(1-p(t_{new}))(p(t_\alpha)r_{\alpha-1,k-1}-\max_{1\leq l\leq k} r_{d,l-1}))\\ &\geq 0. \quad \text{(by Lemma 5.1)}\end{aligned}$$

So (9) still holds on $C(D)\cup\{t_{new}\}$. □

U-Top$k$ queries. Suppose the tuples in D are $t_1, t_2, \ldots$ in the decreasing rank order. Consider a k-vector $T = (t_{m_1}, \ldots, t_{m_k})$. Let $\Pr(T)$ be the probability of T being the top-k tuples in a random possible world. We have

$$\Pr(T) = \prod_{i=1}^{k} p(t_{m_i}) \prod_{i<m_k, t_i \notin T} (1 - p(t_i)).$$

Recall that a U-Top$k$ query returns the vector T with maximum $\Pr(T)$. Let $D_i = \{t_1, \ldots, t_i\}$, and let D_i^p be the subset of D_i containing the k tuples with maximum probabilities in D_i. Define ρ_i as

$$\rho_i = \prod_{t_j \in D_i^p} p(t_j) \prod_{j \le d, t_j \notin D_i^p} (1 - p(t_j)). \quad (11)$$

Definition 5.3.([33]) The *compact set* $C(D)$ for the U-Top$k$ query on an uncertain data set D is $C(D) = D_d$ where d is the smallest such that

$$\max_{k \le i \le d} \rho_i \ge \prod_{1 \le i \le d} \max\{p(t_i), 1 - p(t_i)\}. \quad (12)$$

Lemma 5.2. *$C(D)$ contains at most k tuples with probability greater than $\frac{1}{2}$.*

PROOF. If $C(D)$ contains more than k tuples with probability greater than $\frac{1}{2}$, there exists $d' < d$, and exactly k tuples from $D_{d'}$ have probabilities are greater than $\frac{1}{2}$. We have $\max_{k \le i \le d'} \rho_i \ge \rho_{d'} = \prod_{1 \le i \le d'} \max\{p(t_i), 1 - p(t_i)\}$. This contradicts the fact that d is the smallest such that (12) holds. □

Theorem 5.3. *The compact set defined for U-kRanks queries is self-maintainable with respect to insertions.*

PROOF. We consider the following two cases.

Case 1: $t_{new} \succ_f t_d$. In this case, it is easy to see that $C(D)$ stays unchanged.

Case 2: $t_d \succ_f t_{new}$. Let $D' = C(D) \cup \{t_{new}\}$, also represented as $D' = \{t'_1, \cdots, t'_{d+1}\}$, ordered by rank. Suppose $t'_m = t_{new}$, then for $1 \le i < m$, $t'_i = t_i$; for $m < i \le d$, $t'_{i+1} = t_i$. Let $\hat{p}_i$ denote the k^{th} largest probability in $\{t_1, \cdots, t_i\}$, $\check{p}$ the probability of tuple t_{new}, ρ'_i and ρ_i the value of (11) for the set D' and $C(D)$ respectively. We have

$$\rho'_i = \begin{cases} \rho_i, & i < m \\ \rho_{i-1}(1 - \check{p}), & i \ge m, \check{p} < \hat{p}_{i-1}; \\ \rho_{i-1}(1 - \hat{p}_{i-1})\frac{\check{p}}{\hat{p}_{i-1}} \ge \rho_{i-1}(1 - \check{p}), & i \ge m, \frac{1}{2} > \check{p} \ge \hat{p}_{i-1}; \\ \rho_{i-1}\check{p}\frac{1-\hat{p}_{i-1}}{\hat{p}_{i-1}} \ge \rho_{i-1}\check{p}, & i \ge m, \check{p} \ge \frac{1}{2} > \hat{p}_{i-1}; \\ \rho_{i-1}\check{p}\frac{1-\hat{p}_{i-1}}{\hat{p}_{i-1}}, & i \ge m, \check{p} \ge \hat{p}_{i-1} \ge \frac{1}{2}. \end{cases}$$

We claim that

$$\max_{i=k}^{d+1} \rho'_i \ge \prod_{i=1}^{d+1} \max\{p(t'_i), 1 - p(t'_i)\}. \quad (13)$$

Indeed,

$$\text{RHS of (13)} = \begin{cases} (1 - \check{p}) \prod_{i=1}^{d} \max\{p(t_i), 1 - p(t_i)\}, & \check{p} < \frac{1}{2}; \\ \check{p} \prod_{i=1}^{d} \max\{p(t_i), 1 - p(t_i)\}, & \check{p} \ge \frac{1}{2}. \end{cases}$$

When $p(t_{new}) = \hat{p} \ge \hat{p}_{i-1} \ge \frac{1}{2}$, (13) follows from Lemma 5.2. Otherwise, it follows from the integration of above two equations. So, $C(D') \subseteq C(D) \cup \{t_{new}\}$. □

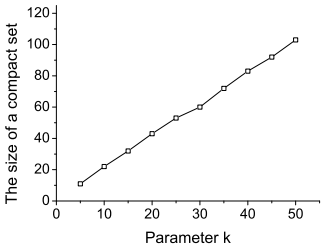
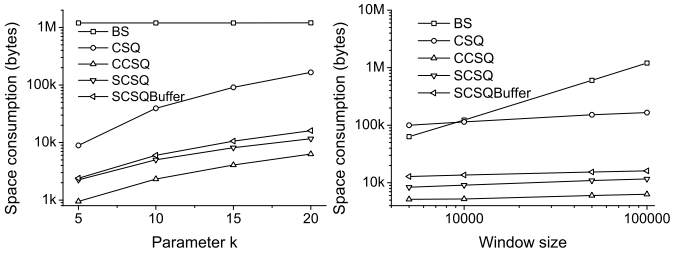


Figure 3: Size of the compact set.



(a) Varying k ($W = 100,000$) (b) varying W (k=20)

Figure 4: Space consumption on synthetic dataset

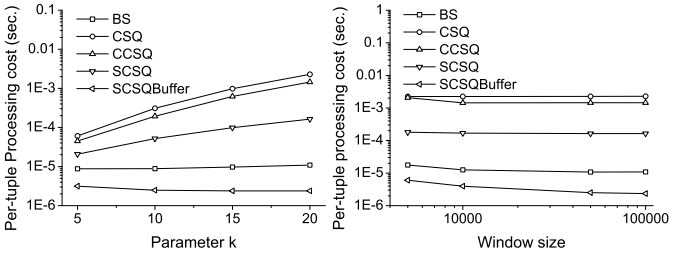
6. EXPERIMENTAL REPORT

In this section, we present an experimental study with both synthetic and real data comparing the five algorithms discussed so far, namely, BS, CSQ, CCSQ, SCSQ, and SCSQ-Buffer. All the algorithms are implemented in C and the experiments are performed on a Linux server with Pentium 4 CPU (2.4GHz) and 1G memory.

Results on synthetic data. We created a synthetic dataset containing 1,000,000 tuples. The rank of each tuple t is randomly generated from 1 to 1,000,000 without replacement and the probability $p(t)$ is uniformly distributed in $(0, 1)$.

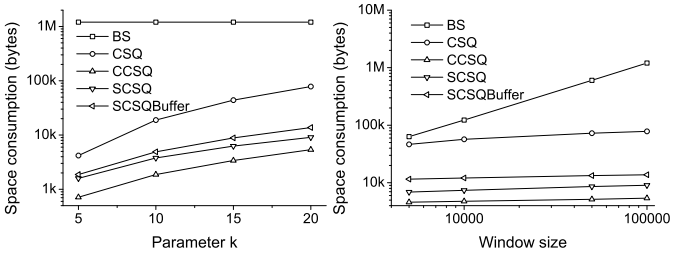
Figure 3 shows the number of tuples in the compact set for the Pk-top$k$ query for this dataset, as k increases. We can see that it is quite small and basically linear in k. This justifies our assumption that H is usually much smaller than the size of the dataset. Note that the previous studies [33, 20] also observed similar behaviors on the size of the compact set.

Next, we feed the dataset in a streaming fashion to each of the synopses and measure their space consumption and processing time. Figure 4 shows the space consumption of the synopses with varying k and varying window size W, respectively. For simplicity, when calculating the space consumption we only counted the tuples and the array r, assuming that each tuple takes 6 bytes and each array entry takes 4 bytes. Keep in mind that, in real applications, the tuples could be much larger as it may contain multiple attributes including long fields like texts. So the sizes for the synopses shown here are only for comparison purposes; the actual sizes will be much larger and application-dependent. The experimental results agree with our theoretical bounds in Table 2 very well: BS is the largest, and its size is dominated by the window size W, irrespective to k. CSQ reduces the size considerably compared with BS, except for very small window sizes. All the other synopses are basically comparable in terms of size, all of which are significantly smaller than CSQ and BS. In general, we observe a space reduction of 2 to 3 orders of magnitude from BS to CCSQ and SCSQ/SCSQ-Buffer on large window sizes.



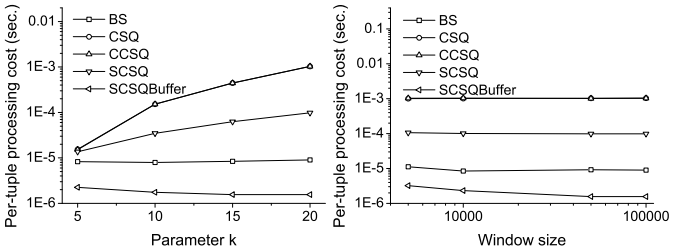
(a) Varying k ($W = 100,000$) (b) Varying W (k=20)

Figure 5: Per-tuple cost on synthetic dataset



(a) Varying k ($W = 100,000$) (b) varying W (k=20)

Figure 6: Space consumption on real dataset



(a) Varying k ($W = 100,000$) (b) varying W (k=20)

Figure 7: Per-tuple processing cost on real dataset

Figure 5 shows the per-tuple processing cost of the five methods. We can observe that CSQ and CCSQ runs slowest, due to their cubic dependency on k (since H is roughly linear in k). SCSQ is better, since the dependency on k is quadratic. BS and SCSQ-Buffer runs the fastest. Interestingly, although they have the same asymptotic bound, we observe that SCSQ-Buffer actually runs even faster than BS. This is a bit counter-intuitive since what BS does for each tuple is very simple. It maintains all the tuples in the window in sorted order (using two balanced binary tree), and simply inserts and deletes tuples in this tree as they arrive and expire. In addition, it rebuilds $C(S_W)$ if it becomes invalid. The latter step is also done in SCSQ-Buffer. The explanation is that although maintaining a balanced binary tree is computationally easy, it is quite memory intensive. When we perform an insertion or a deletion, many nodes in the tree, possibly in different memory locations, are read and written, causing a lot of cache misses. On the other hand, SCSQ-Buffer is much more cache friendly, due to its small memory print and the way it performs the batched updates. Another interesting observation is that the per-tuple processing cost either remains the same or even decreases as the window size increases. The reason is that an incoming tuple has a smaller probability to affect the existing compact sets when the window size is larger, thus saving the computation cost. Similar phenomenon can also be observed in Figure 7(b), 8(d), 9(d), and 10(d).

Results on real data. We used the International Ice Patrol (IIP) Iceberg Sightings Database[3] to examine the efficiency of our synopses in real applications. The (IIP) Iceberg Sightings Database collects information on iceberg activity in North Atlantic to monitor iceberg danger near the Grand Banks of Newfoundland by sighting icebergs, plotting and predicting iceberg drift, and broadcasting all known icebergs to prevent icebergs threatening. In the database, each sighting record contains the date, location, shape, size, number of days drifted, etc. It is crucial to find the icebergs drifting for long periods, so use the number of days drifted as the ranking attribute. Each sighting record in the database contains a confidence level attribute according to the source of sighting, including R/V (radar and visual), VIS (visual only), RAD(radar only), SAT-LOW(low earth orbit satellite), SAT-MED (medium earth orbit satellite), SAT-HIGH (high earth orbit satellite), and EST (estimated, used before 2005). We then converted these six confidence levels to probabilities 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, and 0.4 respectively. We gathered all of records from 1998 to 2007 and result in 44440 records. Based on it, we created a 1,000,000-record data stream by repeatedly selecting records randomly. The experimental results on this real dataset are shown in Figure 6 and 7. We observe very similar results as those on the synthetic data, which demonstrates the robustness of our synopses.

[3]http://nsidc.org/data/g00807.html

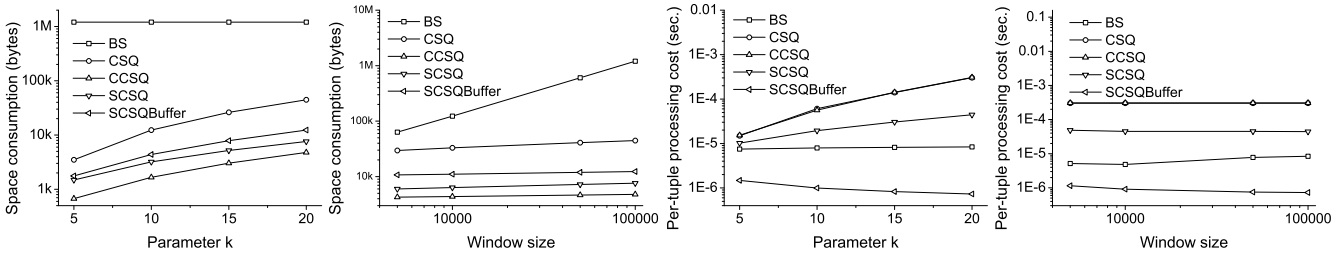
Other top-k queries. Finally, we also implemented the compact sets for the other three top-k definitions: PT-k, U-kRanks, and U-top$k$. We plugged them into our synopses and conducted experiments on the real dataset. The results are shown in Figure 8, 9, and 10. Again, both the space consumption and processing time have very similar behaviors as those on the Pk-top$k$ query, which testifies the generality of our framework.

7. RELATED WORK

Top-k Queries. Top-k queries on a traditional certain dataset have been well studied in the literature. Numerous query processing algorithms have been proposed[16, 26]. The threshold algorithm (TA) [26] is one of the best known algorithms. It assumes that each tuple has several attributes, and the ranking function is a monotone function on these attributes. TA first sorts the tuples by each attribute and then scans the sorted lists in parallel. Each time a new tuple appears, TA looks it up in all lists to calculate its rank. In addition, TA maintains a "stopping value", which acts as a threshold to prune the tuples in the rest of the lists if they cannot have better scores than the threshold.

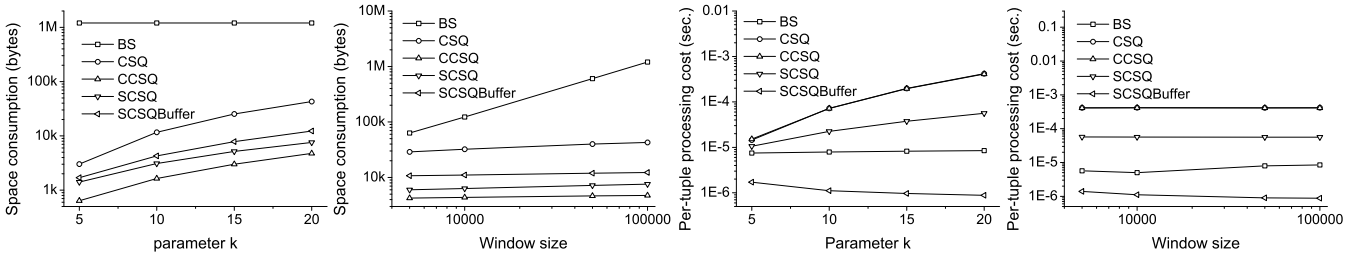
There are many recent development and extensions to top-k queries under different scenarios. Babcock and Olston [4] proposed an algorithm to monitor the top-k most frequent items in a distributed environment. Das et al. [14] use views to answer top-k queries efficiently. Xin et al. [31] remove redundancy in top-k patterns. Xin et al. [32] also apply multidimensional analysis in top-k queries. Hua et al. [19] define the rank of a tuple by the typicality and answer the top-k typicality queries. A very relevant work to ours is the paper by Mouratidis et al. [24], which presents a method to continuously monitor top-k queries over sliding windows. However, same as all of the other works listed above, it only considers certain databases.

Uncertain data management and top-k queries. Uncertain data management [27] has received increasing attention with the emergence of practical applications in domains like sensor networks,



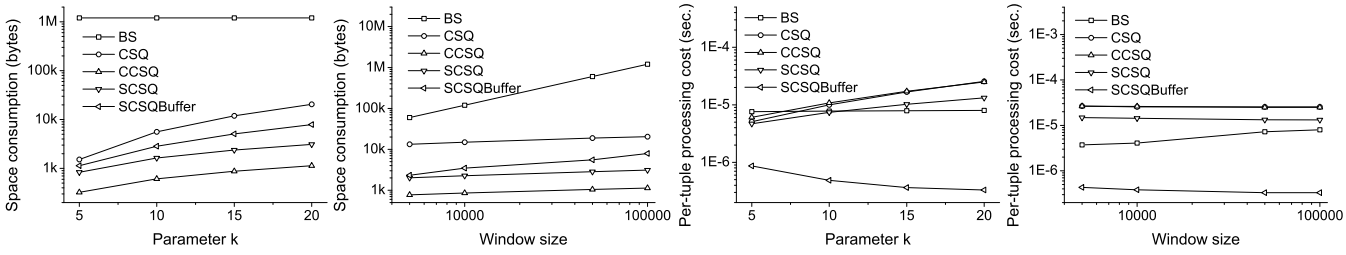
(a) Space used ($W = 10^5$) (b) Space used ($k = 20$) (c) Per-tuple cost ($W = 10^5$) (d) Per-tuple cost ($k = 20$)

Figure 8: PT-k query on real dataset



(a) Space used ($W = 10^5$) (b) Space used ($k = 20$) (c) Per-tuple cost ($W = 10^5$) (d) Per-tuple cost ($k = 20$)

Figure 9: U-kRanks query on real dataset



(a) Space used ($W = 10^5$) (b) Space used ($k = 20$) (c) Per-tuple cost ($W = 10^5$) (d) Per-tuple cost ($k = 20$)

Figure 10: U-top$k$ query on real dataset

data cleaning, and location tracking. The TRIO system [29] introduced different working models to capture data uncertainty at different levels, with an elegant perspective of relating uncertainty with lineage as an emphasis on uncertain data modeling. A good survey on recent uncertain data algorithms is [2].

Cheng et al. [8] provided a general classification of probabilistic queries and evaluation algorithms over uncertain data sets. Different from query answering in traditional data sets, a probabilistic quality estimate was proposed to evaluate the quality of results in probabilistic query answering. Dalvi and Suciu [12] proposed an efficient algorithm to evaluate arbitrary SQL queries on probabilistic databases and rank the results by their probability. Later, they showed in [13] that the complexity of evaluating conjunctive queries on a probabilistic database is either PTIME or #P-complete.

Ŕe et al. [28] gave a solution to answer SQL query over uncertain databases. The idea for their method is to run in parallel several Monte-Carlo simulations, one for each candidate answer, and approximate each probability only to the extent needed to compute the correct top-k answers. However, they are only concerned with the probability of a tuple appearing in the query results, and no ranking function is involved.

There are three definitions proposed so far for uncertain top-k queries based on a ranking function. Soliman et al. [30] first defined two types of such top-k queries, named U-Top$k$ and U-kRanks, and proposed algorithms for each of them. Their algorithms were subsequently improved by Yi et al. [33]. Hua et al. [20, 21] proposed another top-k definition, namely PT-k, and proposed efficient solutions. The Pk-Top$k$ query that we mainly focus in this paper is actually a slight variant of PT-k. But we show how all of the existing three definitions can be plugged into our framework. All the existing works only study the problem of how to answer a "one-shot" top-k query on a static uncertain data set, with the exception of [7], which presents a fully dynamic structure to support *arbitrary* insertions and deletions. However, the structure of [7] has size linear in the data set since the goal there is to allow any tuple to be deleted. In the sliding window model, the "first in, first out" property has allowed us to reduce the space complexity significantly. In addition, [7] is only concerned with U-Topk queries, and the structure is quite complicated and theoretical in nature.

Uncertain data streams. As mentioned in Section 1, there has been a lot of effort in extending the query processing techniques on

static uncertain data to uncertain data streams [1, 9, 10, 11, 22, 23, 34]. Though there are papers on computing statistical aggregates and clustering, there is still no work on top-k queries over uncertain streams. Nevertheless, as we point out in Section 3, it is actually not difficult to extend the existing top-k algorithms to the case of unbounded streams. But in the more meaningful sliding window model, the problem becomes much more difficult. To the best of our knowledge, our paper is the first piece of work on uncertain streaming algorithms in the sliding window model.

8. CONCLUSIONS

Top-k queries are arguably one of the most important types of queries in databases. This paper extends the problem of answering uncertain top-k queries on static datasets to the case of uncertain data streams with sliding windows. We designed both space- and time-efficient synopses to continuously monitor the top-k results, and showed that all the existing top-k definitions can be plugged into our framework. In the present paper, we adopted the simple uncertain data model where each tuple appears with a certain probability independent of other tuples. In future, we are planning to find solutions to cope with more complex uncertain models.

This paper only considers how to cope with sliding-window top-k queries exactly accordingly to the definitions. Another future direction is study the approximate versions of them [21], which possibly allows for more space- and time-efficient solutions.

Acknowledges. The research of Cheqing Jin is supported by the NSF of China under grants No. 60773094 and 60473055, Shanghai Shuguang Program under grant No. 07SG32. The research of Ke Yi is supported by Hong Kong Direct Allocation Grant (DAG 07/08). The research of Lei Chen is supported by RGC HKUST 6119/07E and the National Basic Research Program of China (973 Program) under grant No. 2006CB303000. The research of Jeffrey Xu Yu is supported by RGC HKSAR, China (No. 418206). The research of Xuemin Lin is supported by ARC discovery grants (DP0666428 and DP0881035) and GOOGLE research award.

9. REFERENCES

[1] C. C. Aggarwal and P. S. Yu. A framework for clustering uncertain data streams. In *Proc. of ICDE*, 2008.

[2] C. C. Aggarwal and P. S. Yu. A survey of uncertain data algorithms and applications. In *IBM Research Report. www.charuaggarwal.net/RC24394.pdf*, October, 2007.

[3] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. of ACM STOC*, 1996.

[4] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proc. of SIGMOD*, 2003.

[5] A. Chakrabarti, G. Cormode, and A. McGregor. Robust lower bounds for communication and stream computation. In *Proc. of STOC*, 2008.

[6] A. Chakrabarti, T. Jayram, and M. Pătraşcu. Tight lower bounds for selection in randomly ordered streams. In *Proc. of SODA*, 2008.

[7] J. Chen and K. Yi. Dynamic structures for top-k queries on uncertain data. In *Proc. of ISAAC*, 2007.

[8] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *Proc. of ACM SIGMOD*, 2003.

[9] G. Cormode and M. Garofalakis. Sketching probabilistic data streams. In *Proc. of ACM SIGMOD*, 2007.

[10] G. Cormode, F. Korn, and S. Tirthapura. Exponentially decayed aggregates on data streams. In *Proc. of ICDE*, 2008.

[11] G. Cormode, S. Tirthapura, and B. Xu. Time-decaying sketches for sensor data aggregation. In *Proc. of PODC*, 2007.

[12] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *Proc. of ICDE*, 2004.

[13] N. N. Dalvi and D. Suciu. The dichotomy of conjunctive queries on probabilistic structures. In *Proc. of ACM PODS*, 2007.

[14] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *Proc. of VLDB*, 2006.

[15] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proc. of SODA*, 2002.

[16] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of PODS*, 2001.

[17] L. Golab. *Sliding Window Query Processing over Data Streams*. PhD thesis, University of Waterloo, 2006.

[18] S. Guha and A. McGregor. Approximate quantiles and the order of the stream. In *Proc. of PODS*, 2006.

[19] M. Hua, J. Pei, A. W. C. Fu, X. Lin, and H.-F. Leung. Efficiently answering top-k typicality queries on large databases. In *Proc. of VLDB*, 2007.

[20] M. Hua, J. Pei, W. Zhang, and X. Lin. Efficiently answering probabilistic threshold top-k queries on uncertain data. In *Proc. of ICDE*, 2008.

[21] M. Hua, J. Pei, W. Zhang, and X. Lin. Ranking queries on uncertain data: A probabilistic threshold approach. In *Proc. of SIGMOD*, 2008.

[22] T. Jayram, S. Kale, and E. Vee. Efficient aggregation algorithms for probabilistic data. In *Proc. of SODA*, 2007.

[23] T. Jayram, A. McGregor, S. Muthukrishnan, and E. Vee. Estimating statistical aggregates on probabilistic data streams. In *Proc. of PODS*, 2007.

[24] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *Proc. of ACM SIGMOD*, 2006.

[25] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Foundations & Trends in Theoretical Computer Science, 2005.

[26] S. Nepal and M. V. Ramakrishna. Query processing issues in image(multimedia) databases. In *Proc. of ICDE*, 1999.

[27] D. S. Nilesh and N. Dalvi. Efficient query evaluation on probabilistic databases. In *Proc. of VLDB*, 2004.

[28] C. Ŕe, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *Proc. of ICDE*, 2007.

[29] A. D. Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *Proc. of ICDE*, 2006.

[30] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k query processing in uncertain databases. In *Proc. of ICDE*, 2007.

[31] D. Xin, H. Cheng, X. Yan, and J. Han. Extracting redundancy-aware top-k patterns. In *Proc. of KDD*, 2006.

[32] D. Xin, J. Han, H. Cheng, and X. Li. Answering top-k queries with multi-dimensional selections: the ranking cube approach. In *Proc. of VLDB*, 2006.

[33] K. Yi, F. Li, G. Kollios, and D. Srivastava. Efficient processing of top-k queries in uncertain databases. In *Proc. of ICDE*, 2008.

[34] Q. Zhang, F. Li, and K. Yi. Finding frequent items in probabilistic data. In *Proc. of SIGMOD*, 2008.